View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Purdue E-Pubs

Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1990

Managing Data Synchronization Automatically for Distributed-Memory Architectures

Ko-Yang Wang

Report Number:
90-1043

Wang, Ko-Yang, "Managing Data Synchronization Automatically for Distributed-Memory Architectures" (1990). *Department of Computer Science Technical Reports.* Paper 44.
https://docs.lib.purdue.edu/cstech/44

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

MANAGING DATA SYNCHRONIZATION AUTOMATICALLY FOR DISTRIBUTED-MEMORY ARCHITECTURES

Ko-Yang Wang

CSD-TR-1043
November 1990

Managing Data Synchronization Automatically For Distributed-Memory Architectures

Ko-Yang Wang

Computing About Physical Objects
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907
E-mail address: kyw@cs.purdue.edu

ABSTRACT

In this paper, we discuss problems in parallelizing sequential programs for distributed-memory parallel computers. In particular, we introduce a new program transformation technique called the message consolidation. The message consolidation reduces the communication time for parallel computers that are based on message passing paradigm by decreasing the number of messages on the communication network. Effects of the message consolidation on program performance and conditions for which it is applicable are analyzed. An algorithm for consolidating messages and some heuristics of recognizing the opportunities for applying the transformation are also presented.

November 17, 1990

Managing Data Synchronization Automatically For Distributed-Memory Architectures

Ko-Yang Wang

Computing About Physical Objects
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907
E-mail address: kyw@cs.purdue.edu

1. Data Synchronization for Distributed Memory Systems

Automatic program generation for distributed memory parallel computers, which is a very difficult problem, was largely ignored until the late 1980s'. Nevertheless, the difficulties in programming distributed memory parallel computers make this problem ever more important as the distributed memory architectures such as the NCUBE, NCUBE/2, iPSC/2, iPSC/860, Intel touch stones, etc. become more and more popular. The major difficulty of programming the distributed-memory parallel computers lies in the difficulty of data distribution and communication. One promising approach is to provide the users with a global memory space and let the compilers generate appropriate code for data update. The compilers are responsible for updating and maintaining the consistency of the data in the local memories of the processors. New primitives in the operating system are also being developed to make this task easier and more efficient. This approach is attractive because it allows the users to program the distributed-memory systems in a style close to programming shared-memory computers. However, most existing distributed shared-memory modules require the user to decide the data decomposition and allocation. The performance of these modules is also an important issue for improvement. We do not assume any distributed share memory model here for distributed-memory machines. However, the techniques that we describe for parallelizing sequential programs for distributed-memory machines involve a more general problem. These techniques can be applied to distributed shared memory models to generate optimal code.

For distributed-memory architectures that use the message-passing paradigm, external data references need to be converted into explicit read/write instructions. The simplest approach is to generate a pair of read/write statements for each data dependence and utilize

a set of control libraries that use message passing for control dependences. The problem is that message passing is an expensive operation and the communication overhead and the serialization effects of the read and write operations might destroy any benefits of the parallel execution.

One possible way of reducing the communication cost is to consolidate the messages into longer messages. Consolidating the messages has two apparent effects: decreasing the message passing overheads and increasing the data synchronization delays. Whether two messages should be merged into one depends on the tradeoff between these two factors. Careless message-merging may also generate incorrect results or cause the communication deadlock.

Note that message consolidation has been practiced by parallel programmers in programming distributed parallel computers for a long time but has not yet been utilized in parallel compilers for automatic program optimization of distributed parallel computers.

2. Message Consolidation

Each cross-task data dependence requires a data synchronization point to ensure the correctness of the concurrent execution. Enforcing these data synchronization points would be disastrous on distributed-memory systems or large shared-memory systems that have high network latency because data communication is a very expensive operation on these machines. For example, the cost of sending a 4-byte number to a neighbor on an NCUBE/2 processor costs about 160 microseconds while a floating point multiply costs only 0.35 microsecond. To minimize the cost of data and control synchronization, the compiler has to merge messages into a longer message to save communication cost and overlap communication with computation. In other words, a data synchronization point for multiple data dependence between two tasks is preferred. Unfortunately, merging data synchronization points means delaying the startup time of the data transfer and this decreases the overlapping of the data transmission of the message with the computation at the receiving processor and thus increases the data synchronization cost. It is therefore necessary to derive an algorithm that decides how data synchronization points can be merged beneficially.

Before we discuss the algorithm for performing such tasks, it is necessary to examine some theoretical foundations for the approach. In the following discussion, we assume that the architecture supports read, write and test operations. The read statement sends the message, the write statement receives the message, and the test operation checks if the designated message has arrived. We further assume that the write is non-blocking and the read is blocking, that is, the sending processor can work on other computation after the message is transferred to the underlying network transporting hardware, but the receiving processor will have to wait in the receive statement until the message has arrived. Some architectures provide blocking write statements, but this does not affect the following results (only makes them better since blocking increases the overhead).

In the following discussion, we assume that T_1 and T_2 are two tasks and δ is a flow or output-dependence relation from statement S_1 in T_1 to S_2 in T_2. Let $t(S_1)$ be the execution time of the statements between the first statement in T_1 and S_1 including that of S_1, and $t(S_2)$ be the execution time of the statements between the first statement in T_2 and S_2, excluding that of S_2. and M be the size of the data that causes the data dependence, and transc(M) be the cost of transmitting the data of size M.

Lemma 1. The data synchronization delay in the receiving processor caused by the data dependence δ can be computed by the following formula.

$$delay = \max\left\{0,\ t(S_1) + transc(M) - t'(S_2)\right\}$$

Proof:

The data send by task T_1 will arrive at the task T_2 at time $t(S_1) + transc(M_1)$. If $t(S_1) + transc(M_1) \le t'(S_2)0$ then the data arrives before the statement S_2 is reached, so there is no synchronization delay. On the other hand, if $t(S_1) + transc(M_1) > t'(S_2)$ then the processor that runs task T_2 will have to be idle until the data arrives, so the idle time is $t(S_1) + transc(M_1) - t'(S_2)$. Combining the two cases, the synchronization delay caused by the dependence δ is then $delay = \max\left\{0,\ t(S_1) + transc(M) - t'(S_2)\right\}$.

QED.

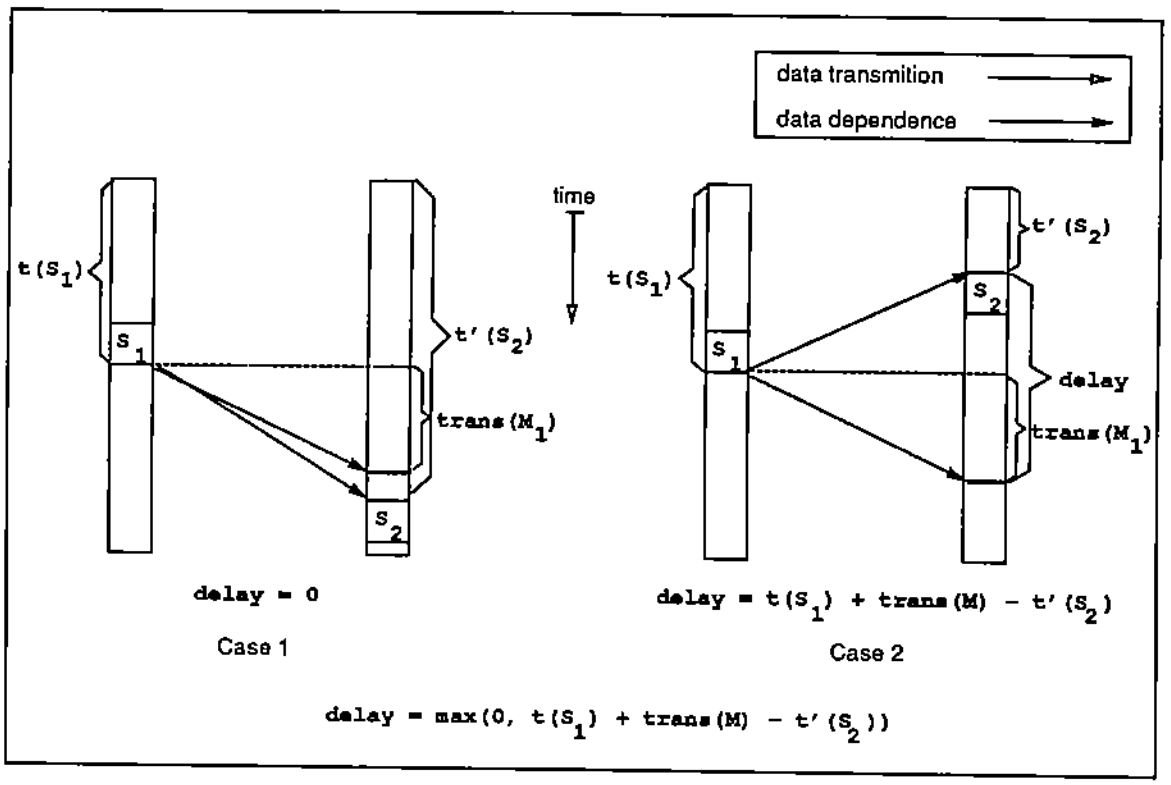

Figure 1. The synchronization delay cause by the inter-task communication.

For the case of merging multiple data synchronization points of multiple dependence into one, we introduce a new dependence relation called the *data dependence cluster*.

Definition 1. A *data dependence cluster* is a quadruple $(\Omega, \Delta, S_1, S_2)$ where Ω is a set of data dependences from task T_1 to task T_2, Δ is the union of the data involved in the dependences in Ω, S_1 is a statement in T_1 that must be executed before the data can be sent to T_2, and S_2 is a statement in T_2 where the data involved in the dependences in Ω must arrive or the execution of T_2 will be blocked.

The data dependence cluster is a generalization of the data dependence. A data dependence δ involving the data d from statement S_1 to statement S_2 defines a data dependence cluster: $(\{\delta\}, \{d\}, S_1, S_2)$. And the effect of the data dependence cluster on the performance of the program is the same as that of the data dependence.

Corollary 1.1 The data synchronization delay for a data dependence cluster that contains one dependence relation is the same as the data synchronization delay caused by the

dependence.

Many operations can be defined on the data dependence cluster, but the operation that we are interested here is the union (called merge below) operation. We define the union of two data dependence clusters $(\Omega^a, \Delta^a, S^a{}_1, S^a{}_2)$ and $(\Omega^b, \Delta^b, S^b{}_1, S^b{}_2)$ to be $(\Omega^a \bigcup \Omega^b, \Delta^a \bigcup \Delta^b, S'_1, S'_2)$, where S'_1 is $S^a{}_1$ or $S^b{}_1$, depending on which statement lexicographically behind the other. and S'_2 is $S^a{}_2$ or $S^b{}_2$, depending on which statement lexicographically in front of the other.

Definition 2. Two data dependence clusters Ψ^a and Ψ^b are called mergeable if merging the two dependence clusters will not change if there is no dependence δ from statement S_l to S_k such that S_l is between S_{a_2} and S_{b_2} in T_2 and S_k is between statements S_{a_1} and S_{b_1} in T_1.

Trying to merge two data dependence clusters that are not mergeable would violate the data dependence relation by moving the source of a data dependence beyond a statement that depends on it. Furthermore, this would cause the two tasks to deadlock since task T_1 would have to wait for data from T_2 before it could process the statement S_k and task T_2 will have to wait for data from T_1 before it can process statement S_l, thus the deadlock.

The data dependence cluster can be used to guide the code generation for distributed-memory architectures. For example, a write statement is generated after statement S_1 which sends the data in Δ and a read statement is generated in front of the statement S_2 to receive the data. Note that the sets of data dependences in the data dependence clusters are mutually exclusive and that the data dependence clusters form a partition of the set of the data dependences. A cluster $Cluster_1$ is defined to be '<' $Cluster_2$ if all statements in T_1 that are involved in the dependence relations in the $Cluster_1$ are in front of those in $Cluster_2$. This defines a partial order of the clusters. Two clusters are said to be *adjacent* to each other if there are no clusters between them.

Lemma 2. If there are two data dependences, δ^1 and δ^2, from tasks T_1 to T_2; δ^1 is from statements S_1 in T_1 to S_2 in T_2 and δ^2 is from statements S_3 in T_1 to S_4 in T_2. Let $t(S_i)$ be the execution time of the statements between the first statement in T_1 and S_i including that of S_i, and $t'(S_j)$ be the execution time of the statements between the first statement in

T_2 and S_j, excluding that of S_j. Let M^i be the size of the data that causes the data dependence δ^i, and transc(M^i) be the cost of transmitting the data of size M^i. Then, the delay in T_2 caused by the two data dependences is the maximum of the two delays. In other words, the delay for task T_2 is:

$$delay = \max\left\{ t(S_1) + transc(M^1) - t'(S_2),\ t(S_3) + transc(M^2) - t'(S_4) \right\}$$

$$= \max\left\{ delay^1,\ delay^2 \right\}$$

Proof:

By Lemma 1, we know that the delay caused by dependence δ^1 is

$$delay^1 = \max\left\{ 0,\ t(S_1) + transc(M^1) - t'(S_2) \right\}$$

The problem may be divided into two cases based on the order of statements S_2 and S_4.

Case 1. $t'(S_2) \le t'(S_4)$ (as shown in figure 2):

Due to the dependence δ^1, a delay of $delay^1$ has to be inserted before the statement S_2; as a result, every statement in task T_2 after S_2 is delayed by this amount of time. So the statement S_4 will be reached at time $t'(S_4) + delay^1$, and by Lemma 1, the delay caused by the dependence δ^2 becomes

$$delay^{2'} = \max\left\{ 0,\ t(S_3) + transc(M^2) - (t'(S_4) + delay^1) \right\}$$

$$= \max\left\{ 0,\ (t(S_3) + transc(M^2) - t'(S_4)) - delay^1 \right\}$$

$$= \max\left\{ 0,\ delay^2 - delay^1 \right\}$$

And the delay for both dependences is

$$delay = delay^1 + delay^{2'} = delay^1 + \max\left\{ 0,\ delay^2 - delay^1 \right\}$$

$$= \max\left\{ delay^1,\ delay^2 \right\}$$

Case 2. $t'(S_2) > t'(S_4)$ (as shown in figure 3):

Since the statement S_4 is in front of the statement S_2, the $delay^2$ that is inserted in front of statement S^4 also delays the time statement S_2 gets executed. So the new delay for statement S_2 is:

$$delay^{1'} = \max\left\{ 0,\ t(S_1) + transc(M^1) - (t'(S_2) + delay^2) \right\}$$

$$= \max\left\{ 0,\ (t(S_1) + transc(M^1) - (t'(S_2)) - delay^2 \right\}$$

$$= \max\left\{ 0,\ delay^1 - delay^2 \right\}.$$

This implies that the synchronization delay caused by dependences δ^1 and δ^2 is

$$delay = delay^2 + delay^{1'} = delay^2 + \max\left\{ 0,\ delay^1 - delay^2 \right\}$$

$$= \max\left\{ delay^2,\ delay^1 \right\}$$

This concludes the proof.

QED.

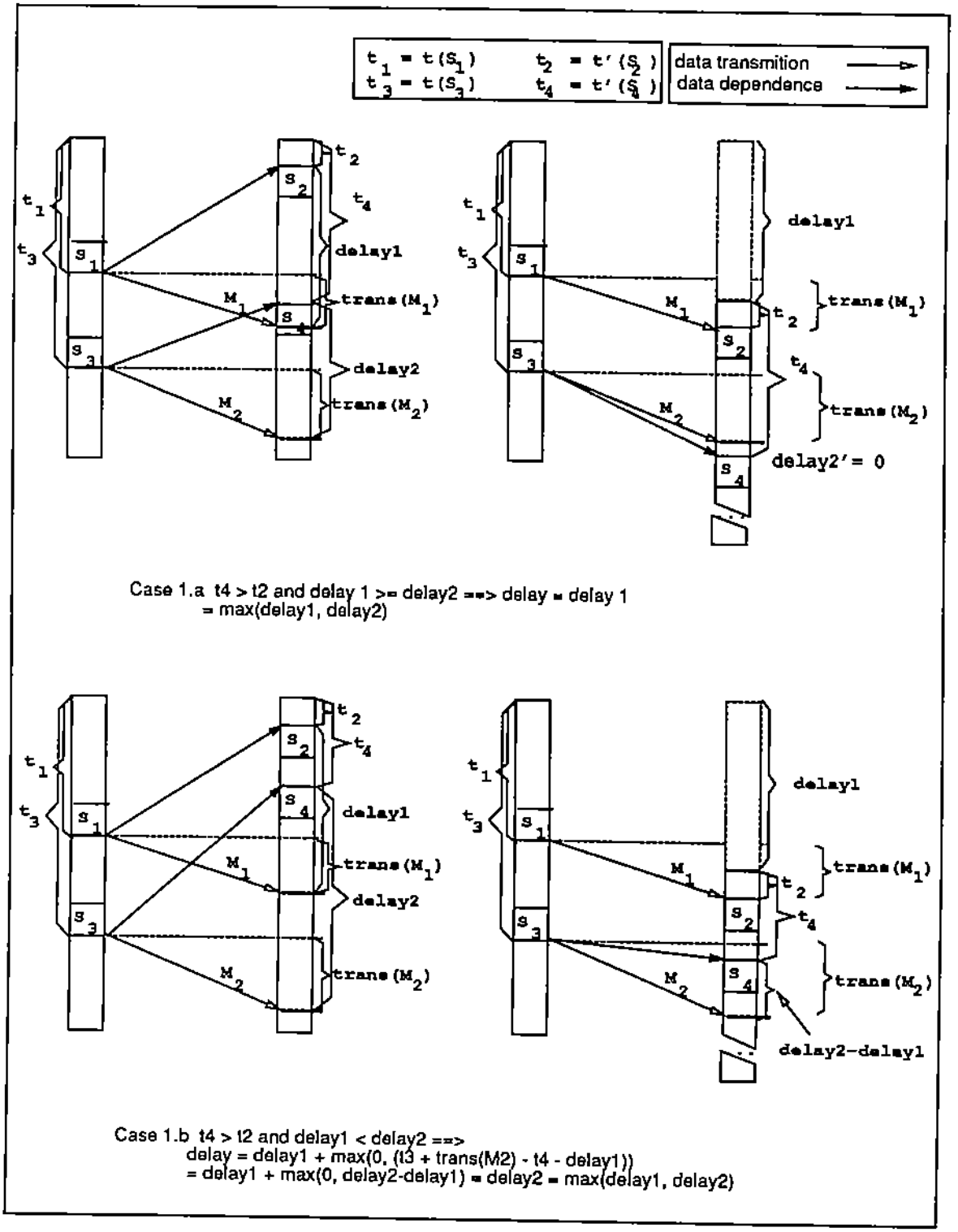

Figure 2. The synchronization-delay caused by two data dependences between tasks T_1 and T_2 (statement S_2 is in front of S_4).

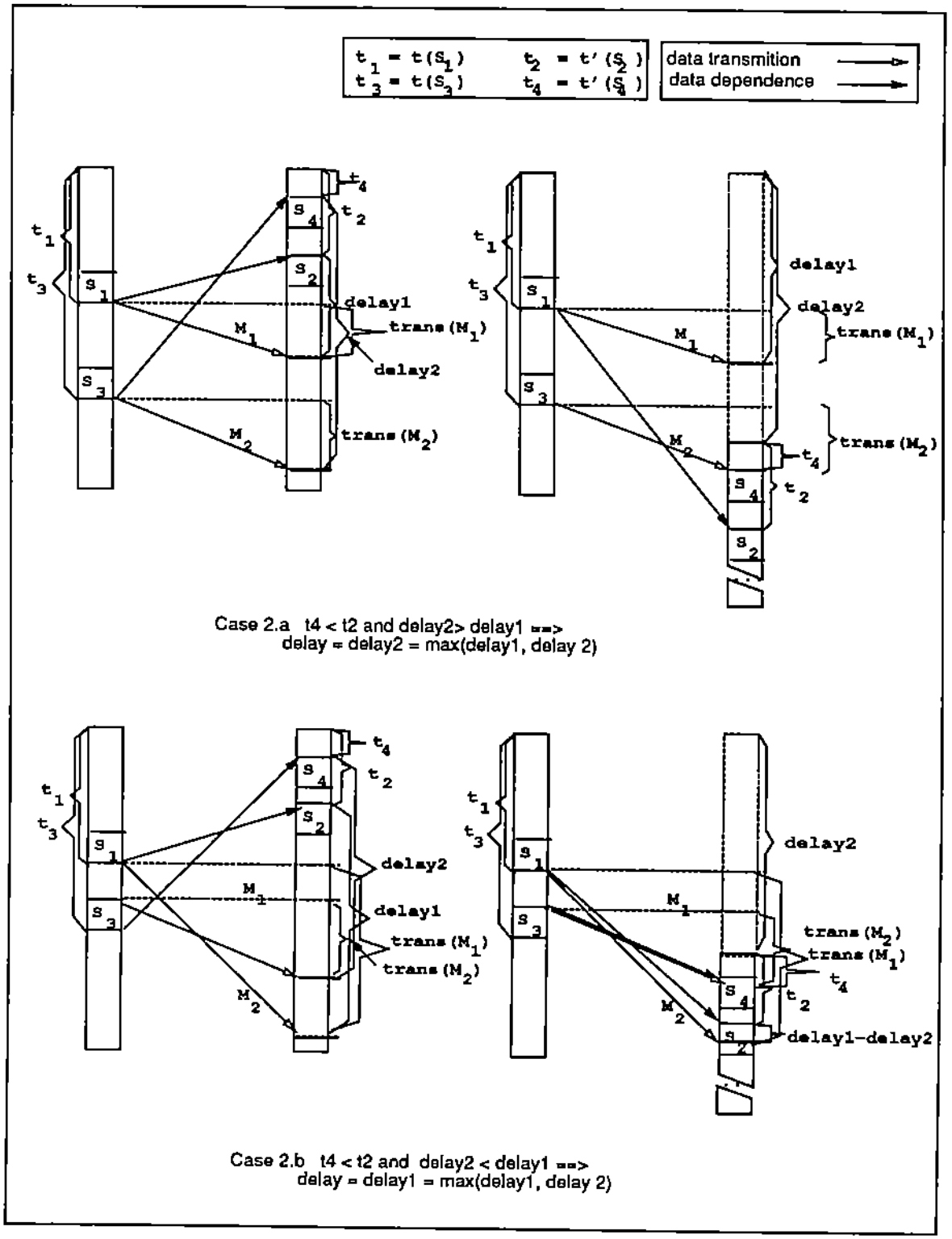

Figure 3. The synchronization-delay caused by the two data dependences between tasks T_1 and T_2 (statement S_4 is in front of S_2).

Note that case 2.b in figure 3 can happen on architectures that support alternative routing when the message size M^1 is very large. For other machines that use fixed routing between two processors, the first message may block the second message so this case is not possible. The lemma still holds because the transmission time for sending the second message will be much longer thus causing a longer delay.

A direct generalization of the above lemma leads to the following lemma.

Lemma 3. When there is more than one data dependence relation between two tasks, the synchronization delay in task T_2 is determined by the dependence that causes the longest delay. That is, if there are m data dependences from tasks T_1 to T_2 where the i-th dependence δ^i is from statements S_f^i to S_t^i, and the i-th delay, $delay^i$, is caused by δ^i, then the delay for task T_2 caused by dependences from task T_1 is:

$$delay = \max_{i=1}^{m} \left\{ t(S_f^i) + transc(M^i) - t'(S_t^i) \right\}$$

$$= \max_{i=1}^{m} \left\{ delay^i \right\}$$

Proof:

The lemma can be proved by induction. The base case is when there are two data dependences and is proven in Lemma 2. Assuming that the lemma is true for any m dependences where m < n, we now proceed to prove for the case of n dependences. Assuming that the dependences are ordered by the order of the statements in task T_2. For the first $n-1$ dependences between the two tasks, assume that $delay^k$ is the largest delay in the delays caused by the dependences, then by the assumption, the combined delay for the first $n-1$ dependences is $delay^k$. Now consider the dependence δ^n, the statement S_t^n is delayed for $delay^k$ by the previous $n-1$ dependences. The delay for the n-th dependence is then:

$$delay^{n'} = \max \left\{ 0,\ t(S_f^n) + transc(M^n) - (t'(S_t^n) + delay^k) \right\}$$

$$= \max \left\{ 0,\ delay^n - delay^k \right\}$$

So the overall delay of all n dependences is

$$delay = delay^k + \max\left\{0,\ delay^n - delay^k\right\}$$

$$= \max\left\{delay^k,\ delay^n\right\}$$

$$= \max\left\{\max_{i=0}^{n-1}\left\{delay^i\right\},\ delay^n\right\}$$

$$= \max_{i=0}^{n}\left\{delay^i\right\}$$

This completes the proof.

QED.

The following formula determines the new delay in task T_2 when the first message is merged into the second.

Lemma 4. If the two messages as defined in lemma 2 are merged into one, then the synchronization delay for task T_2 becomes:

$$delay = \max(0,\ t(S_3) + transc(M^1 + M^2) - \min(t'(S_2),\ t'(S_4))$$

Proof:

When the first message is merged into the second message, the data dependence from statement S_1 into statement S_2 is changed into a dependence from statement S_3 into statement S_2, and the size of the data to be sent from task T_1 to task T_2 is increased into M^1+M^2. At time $t(S_3) + transc(M^1 + M^2)$ the message will be available on task T_2 so the delay is this time minus the time the first statement involved in the dependences is reached which is $\min(t'(S_2),\ t'(S_4))$. This proves the formula.

QED.

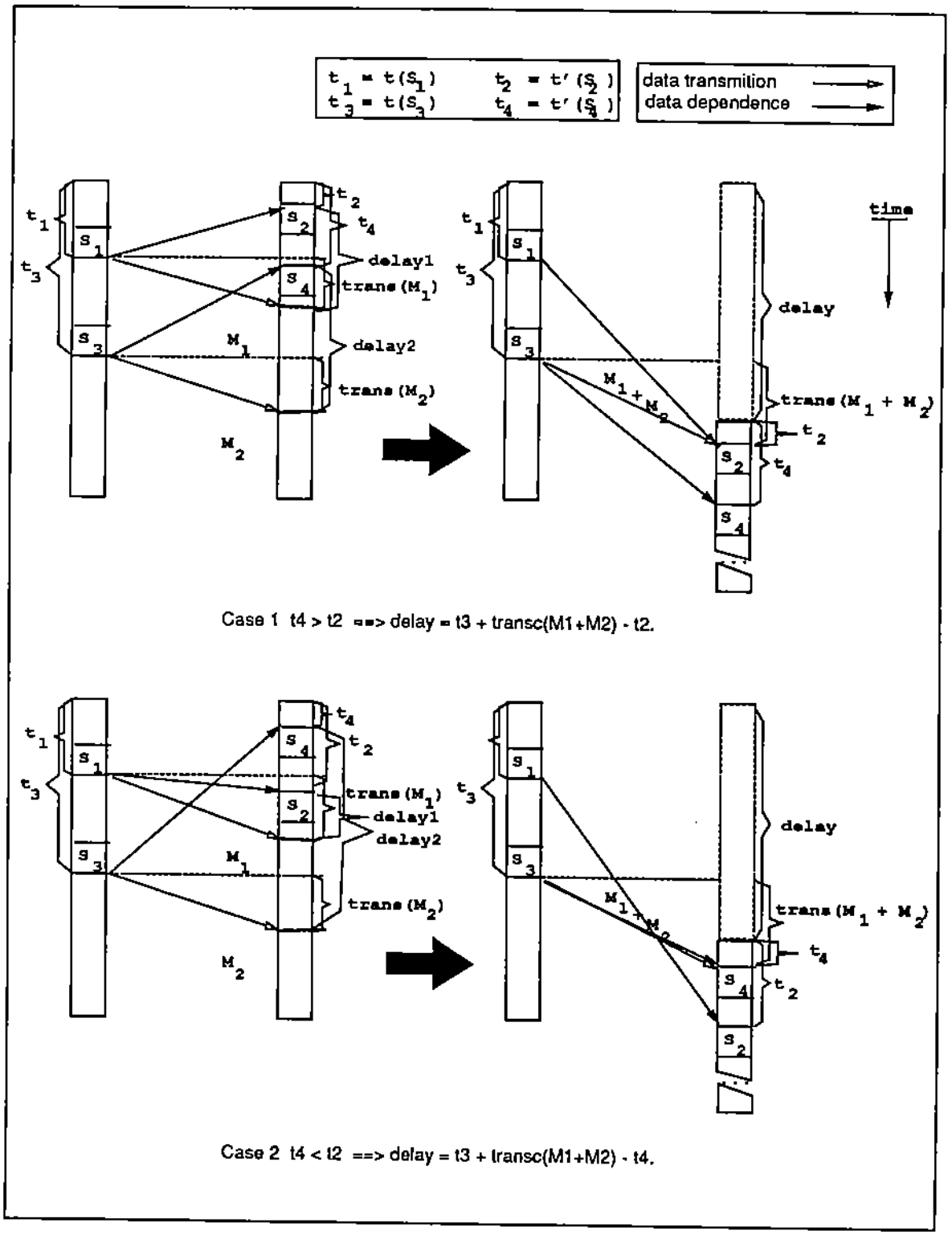

Figure 4. The synchronization delay for merging two messages into one.

The drawback of merging two messages is that the corresponding statements in task T_2 have to wait for the message to arrive, which might increase the data synchronization time.

On most distributed systems, long messages are preferred over short messages for two reasons. First, the overhead for message startup is high; second, the hot-spot effects of having many messages going to certain hot-spots can degrade the network performance in a dramatic way. The message transmission time *transc* () is a function of the size of the message and the distance between the two processors.

$$transc\,(N,\ hops) = \alpha(hops) + \beta(hops)*N + hot^{delay}.$$

Where $\alpha(hops)$ is the message startup time and $\beta(hops)$ is the unit cost for data transmission (both are functions with the distance between the two processors as a parameter), and hot^{delay} is the hot-spot delay caused by message collision (see [Wang90a] for more detail). Let $\phi(L)$ be the function representing the estimated decrease in the data transmitting delay when one message was removed from a hot-spot with L messages.

Lemma 5. For the same condition as in lemma 2, the extra cost C^m of merging the two messages is $C^m = delay - max(delay^1, delay^2)$. If $\max(delay^1, delay^2) > 0$ then $C^m = \Gamma - \phi(L)$ where Γ is defined to be:

$t(S_3) - t(S_1) + \beta(hops) * M^2$	if $t'(S_2) \le t'(S_4)$ and $delay^1 \ge delay^2$
$t'(S_4) - t'(S_2) + \beta(hops) * M^1$	if $t'(S_2) \le t'(S_4)$ and $delay^1 < delay^2$
$t(S_3) - t(S_1) + t'(S_4) - t'(S_2) + \beta(hops) * M^1$	if $t'(S_2) > t'(S_4)$ and $delay^2 \ge delay^2$
$\beta(hops) * M^1$	if $t'(S_2) > t'(S_4)$ and $delay^1 < delay^2$

Proof:

By definition $C^m = delay - \max(delay^1, delay^2)$.

If $t'(S_4) \ge t'(S_2)$ and $delay^1 \ge delay^2$ (as in figure 2 case 1.a) then the delay caused by the two data dependences is $delay^1$. Since $delay^1 > 0$, the extra cost of merging the two messages is

$$C^m = (t(S_3) + transc\,(M^1 + M^2) - t'(S_2)) - (t(S_1) + transc\,(M^1) - t'(S_2))$$

$$= (t(S_3) - t(S_1)) + (transc\,(M^1 + M^2) - transc\,(M^1))$$

$$= t(S_3) - t(S_1) + \beta(hops) * M^2 - \phi(L).$$

If $t'(S_4) \geq t'(S_2)$ and $delay^1 < delay^2$ (as in figure 2 case 1.b) then the delay caused by the two data dependences is $delay^2$. Since $delay^2 > 0$, the extra cost of merging the two messages is then

$$C^m = (t(S_3) + transc(M^1 + M^2) - t'(S_2)) - (t(S_3) + transc(M^2) - t'(S_4))$$
$$= (t'(S_4) - t'(S_2)) + (transc(M^1 + M^2) - transc(M^2))$$
$$= t'(S_4) - t'(S_2) + \beta(hops) * M^1 - \phi(L).$$

If $t'(S_4) < t'(S_2)$ and $delay^1 \geq delay^2$ (as in figure 3 case 2.a) then the delay caused by the two data dependences is $delay^1$. Since $delay^1 > 0$, the extra cost of merging the two messages is

$$C^m = (t(S_3) + transc(M^1 + M^2) - t'(S_4)) - (t(S_1) + transc(M^1) - t'(S_2))$$
$$= (t(S_3) - t(S_1)) - (t'(S_4) - t'(S_2)) + (transc(M^1 + M^2) - transc(M^1))$$
$$= (t(S_3) - t(S_1)) - (t'(S_4) - t'(S_2)) + \beta(hops) * M^2 - \phi(L).$$

If $t'(S_4) < t'(S_2)$ and $delay^1 < delay^2$ (as in figure 3 case 2.b) then the delay caused by the two data dependences is $delay^2$. Since $delay^2 > 0$, the extra cost of merging the two messages is

$$C^m = (t(S_3) + transc(M^1 + M^2) - t'(S_4)) - (t(S_3) + transc(M^2) - t'(S_4))$$
$$= (transc(M^1 + M^2) - transc(M^2))$$
$$= \beta(hops) * M^1 - \phi(L).$$

Since $\Gamma = C^m + \phi(L)$, this complete the proof.

QED.

We separate the function C^m into two functions Γ and $\phi(L)$ because the hot-spot effect and the network transmission delay are very sensitive to the global state of the network; and $\phi(L)$ is very difficult to predicate correctly. Also, when the message-passing pattern between tasks shows that the messages are spread out, the term $\phi(L)$ can be dropped altogether. If the program focus is a parallel loop, heuristics for estimating the network transmitting delay can be derived from the pattern of the communication between

tasks. Since the hot-spots usually involve having many messages sent to one processor, one heuristic is concerned only with the hot-spots occurring inside parallel loops. For messages not in any loop we simply ignore the saving in data transmission time and drop the term $\phi(L)$ from computation of C^m. This is because the messages outside loops are more random (so are more spread out) and not as predictable as those inside loops. This assumption is very conservative since decreasing the number of messages always has positive effects on the network congestion. But in general, parallel loops have much higher chances to produce hot-spots than sequence of statements. Only when the optimization degree is very high should we worry about the function $\phi(L)$ when merging the messages.

As can be seen in the last lemma, merging the messages together will increase the data synchronization overhead in task T_2. Why is it the case, then, that on most distributed memory systems merging the messages would speed up the computation instead of slow it down? This is because merging two messages into one has three effects:

1. It increases the synchronization delay of the receiving processor that runs task T_2 as described in lemma 5 by delaying the sender.
2. It decreases the execution time of the processor running task T_1 by C^{send} (overhead of sending a message).
3. It saves data transmitting time by easing the hop-spot effects. When there are message hot-spots [PfNo85], the network congestion will cause a slow down to the whole network and greatly increase the data transmission time. Merging the messages can decrease the number of messages simultaneously in the network and relieve the hot-spot congestion.

The above three effects apply to three different entities; although there is no clear-cut method for estimating the combined effects on the overall performance of the program, Lemma 5 can be used as a guideline for deciding when to merge two messages. Messages should be merged only when the overhead for sending a message and the decrease in hot-spot effect justifies the extra data synchronization cost caused by delaying the sending of the first data.

Heuristic 4. Assuming C^m is the overhead in task T_2 for merging the two messages, C^{send} is the overhead for sending a message, and $hot^{delay}(M)$ is the data transmission

delay caused by a hot-spot when there are M messages involved in the hot-spot, then the messages can be merged when the following condition is satisfied:

$$C^m = delay - \max(delay^1, delay^2) \leq C^{send}$$

Where C^m is defined in Lemma 5. By the same lemma, the above condition is equivalent to $\Gamma \leq C^{send} + \phi(L)$.

The above heuristic assumes that the delays in different tasks have the same effects on the overall performance of the program. This assumption is again conservative. Although changes in the execution time of any task would affect all tasks that interact with it, it is more than likely that the overhead in task T_2 can be masked by overlapping the communication with computation. On the other hand, the saving in eliminating one message sending will directly decrease the execution time of T_1.

Based on the heuristic 4 and Lemma 5, a heuristic-guided algorithm for deciding when messages can be profitably merged is derived. The problem of data synchronization can be defined as a problem of partitioning the data dependences into data dependence clusters. Initially, we assume that each data dependence relation forms a data dependence cluster by itself. We then proceed to merge (union) the clusters into larger clusters until the merge is no longer beneficial. This algorithm is applied to each pair of tasks $\{T_1, T_2\}$ that have data dependences from task T_1 to task T_2.

Algorithm 1. Message merging.

For each pair of tasks T_1 and T_2 with data dependence from T_1 to T_2 do.

1. Each data dependence forms a data dependence cluster of its own.
2. For each data dependence cluster Ψ^i do

    for each data dependence cluster Ψ^j do

        If Ψ^i and Ψ^j are mergeable then

            calculate C^m, the cost of merging Ψ^i and its adjacent data dependence cluster Ψ^j, and set $B^{i,j} = C^{send} - C^m$

        else

            set $B^{i,j}$ to be $-\infty$.

end if

end for

3. Sort the pair of data dependence clusters (Ψ^i, Ψ^j) in a heap based on $B^{i,j}$

4. While the minimum B of those of all data dependence cluster pairs is positive do

merge Ψ^i and Ψ^j into $\Psi^{i'}$

recompute the value $B^{i',j}$ for all Ψ^j and adjust the heap.

end while.

END.

This algorithm has the complexity of $\Omega(n^2 * \log(n))$ where n is the number of cross-task data dependences. This is because that in step 2. the cost calculation was repeated $(n-1)^2$ times. In step 3, the cost of sorting n^2 numbers is $\Omega(n^2 * \log(n))$. There will be at most $n-1$ merges in step 4. So the overall complexity is $\Omega(n^2 * \log(n))$.

This algorithm is heuristic-oriented because we did not try hard to compute the hot-spot effect accurately. The function ϕ is based on the evaluation function for hot-spot effects and is ignored (assume it has value 0) for cases outside loops. To improve the algorithm, we apply a heuristic that merges only adjacent data dependence clusters. This heuristic make good sense, because the further the statements $S^a{}_1$ and $S^b{}_1$ are apart, the longer the synchronization delay on T_2 will be.

Algorithm 2. Message merging (with the heuristic that merges only adjacent dependences).

For each pair of tasks T_1 and T_2 that have data dependence from T_1 to T_2 do.

1. Each data dependence forms a data dependence cluster of its own.

2. For each data dependence cluster Ψ^i do

Calculate C^m, the cost of merging Ψ^i and its adjacent data dependence cluster Ψ^{i+1} (as defined in Lemma 5) if they are mergeable. And $B^i = C^{send} - C^m$

end for

3. Sort the data dependence clusters in a heap based on B^i

4. While the minimum B^i of those of all data dependence clusters is positive do

   merge Ψ^i and its adjacent data dependence cluster Ψ^{i+1} into $\Psi^{i'}$

   Compute the value B' for $\Psi^{i'}$ and adjust the heap.

   end while.

   END.

Lemma 6. The above algorithm is $\Omega(n * \log(n))$ with respect to the number of dependences n.

Proof

Initially there are n dependence clusters, and after each merge there is one less data dependence cluster. So in a worst case, the algorithm can execute the while loop in 4 at most $n-1$ times. In step 3, sorting n numbers takes $\Omega(n * \log(n))$, so the overall complexity of the algorithm is $(OMEGA(n * \log(n))$.

QED.

Note that if we omit the sorting in step 3 in algorithm 2 and merge the data dependence clusters Ψ^i and Ψ^{i+1} if B^i is positive, then, the algorithm 2 is linear. The drawback of these heuristics is that we lost the optimal claim of the algorithm 1 when speeding up the algorithm.

When loops are parallelized, it is usually done by blocking the loops into a parallel loop and a sequential inner loop. The external data references inside the sequential loop often define a very regular pattern of accesses. Thus the message consolidation algorithm can take advantage of this regularity and does not need to unroll the loop to consolidate the messages. Instead, the algorithm can work on the statements inside the loop and at the end of the statement blocks, assumes the loop wraps around once and generalizes the result to all loop instances. This implies that the complexity of the algorithm for parallel loops is the number of cross-task dependences in the inner-loops.

Statement reordering can be used to move the definition of the data that is the source of a cross-task dependence to as early as possible and move the use of the cross-task data to as late as possible. This has the effect of minimizing the synchronization delay.

The transformation *array reshaping* [Wang90b] can be used to condense the size of the data to be moved across the network to further decrease the data transmitting time. The dependence graph is adjusted so that the data references in the receiving task depend on the local variables that hold the arriving messages instead of the original variables. This keeps the dependence graph in a consistent state.

The message consolidation and other program transformations are implemented in an intelligent parallel compiler for different parallel computers [Wang90c] that is under development. The compiler utilizes the feature-directed program optimization model [WaGa89] that optimizes user programs under the guidance of an expert system based on the features of programs and architectures [Wang89].

3. Related Works

There has been some recent research in automatic code generation for distributed memory computers. In [Koel90], Koelbel described a method that uses compile and run-time dependence tests to generate code for forall loops in the Kali compiler. However, this technique cannot be readily extended to other languages such as Fortran or C because the technique assumes the functional semantic of the Kali forall loops - all data are assumed to be copied in at the beginning of the loop and copied out at the end of the loop. This means that there is no inter-processor communication inside the loops.

In [CaKe88], Callahan and Kennedy discussed a language for programming distributed-memory computers which consists of a sequential, shared-memory base language extended with directives for specifying the distribution of the program data elements. A LOAD-STORE mechanism was specified as intermediate steps and merging load statements were also suggested. Our work in this paper is similar to their work but is more general and go a big step forward in assuming that the input language is sequential and analyze conditions for merging messages and effects of message consolidation on the performance.

Gerndt [Gern89] presented a framework for automatic code generation for distributed-memory machines that is similar to the one reported in [CaKe88]. This framework also supports the concept of data overlapping between local copies of different processors. But he didn't consider merging the messages. Our message consolation can be

used to optimize the performance of the program and can actually be applied to the above two systems.

4. Conclusions

The major problems in parallelizing sequential programs for distributed-memory parallel computers lie in distributing data into local memories and converting data dependences into messages. In this paper, we discussed the problem of reducing data synchronization by consolidating data dependences and thus messages. We introduced a special kind of data dependence called the data dependence cluster. We analyzed performance of merging data dependence clusters and messages based on the estimated parallel execution time of the program. The message consolidation techniques introduced in this paper can reduce the communication overhead for parallel computers significantly when they are applied appropriately. The algorithm for message consolidation that we presented here utilizes a heuristic-guided performance prediction model to decide whether two messages can be beneficially merged. The algorithm is optimal in the sense that it minimizes the performance predicated by the model.

5. REFERENCES

[BuCy86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *SIGPLAN symposium on Compiler Construction,* 1986, 613-641.

[CaKe88] C.D. Callahan, and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing,* Vol 2., 1988, 151-169.

[FeOtWa83] J. Ferrante, K. Ottenstein, J. Warren, "The Program Dependence Graph and Its Uses in Optimization," IBM Technical Report RC 10543, Aug. 1983.

[PfNo85] G. Pfister, V.A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks," *Proc. of the 1985 International Conference on Parallel Processing,* 1985, 790-797.

[Gern89] H.M. Gerndt, "Automatic Parallelization for Distributed-Memory Multiprocessing Systems," Ph.D. thesis, University Bonn, Dec. 1989.

[Koel90] C. Koelbel, "Compiling Programs For Nonshared Memory Machines," Ph.D. thesis, Department of Computer Science, Purdue University, Dec. 1990. Also available as technical report CSD-TR-1037, 1990.

[Wang89] K. Wang, "Machine Knowledge Representation and Manipulation For Parallel Compilers," Technical Report CSD-TR-843, Department of Computer Sciences, Purdue University, Jan. 1989.

[WaGa89] K. Wang and D. Gannon, "Applying AI Techniques to Program Optimizations For Parallel Computers," in K. Hwang and D. DeGroot, editors, *Parallel Processing for Supercomputers and Artificial Intelligence,* McGraw-Hill, 1989, 441-485.

[Wang90a] K. Wang, "A Performance Prediction Model For Parallel Compilers," Tech. Report, CSD-TR-1041, CER-90-43, Department of Computer Science, Purdue University, Nov. 1990.

[Wang90b] K. Wang, "Array Reshaping - A Mechanism for Optimizing Array Storage On Parallel Architectures," Tech. Report, CSD-TR-1042, CER-90-44, Department of Computer Science, Purdue University, Nov. 1990.

[Wang90c] K. Wang, "A framework For Intelligent Parallel Compilers," Tech. Report, CSD-TR-1044, Department of Computer Science, Purdue University, Nov. 1990.